Towards explicit rewrite rules in the λΠ-calculus modulo

Ronan Saillard

MINES ParisTech
Fontainebleau, France
ronan.saillard@mines-paristech.fr

Abstract

This paper provides a new presentation of the λΠ-calculus modulo where the addition of rewrite rules is made explicit. The λΠ-calculus modulo is a variant of the λ-calculus with dependent types where β-reduction is extended with user-defined rewrite rules. Its expressiveness makes it suitable to serve as an output language for theorem provers, certified development tools or proof assistants. Addition of rewrite rules becomes an iterative process and rules previously added can be used to type new rules. We also discuss the condition rewrite rules must satisfy in order to preserve the Subject Reduction property and we give a criterion weaker than the usual one. Finally we describe the new version of *Dedukti*, a type-checker for the λΠ-calculus modulo for which we assess its efficiency in comparison with *Coq*, *Twelf* and *Maude*.

1 Introduction

Since the pioneering works on *Automath* [6], the λΠ-calculus (*a.k.a.* LF or λP) has been used to specify logics and programming languages [13]. In [10], Dowek and Cousineau suggested that extending this language with rewrite rules would greatly increase its expressiveness. The resulting language, called λΠ-calculus modulo, has since been used as an output language for theorem provers [7] [11] as well as for certified development tools [8] or proof assistants [2].

In this paper we provide a new presentation of the λΠ-calculus modulo that makes explicit the addition in the system of rewrite rules and we discuss the typing conditions they must satisfy. We also describe our new version of *Dedukti*, a type-checker for the λΠ-calculus modulo, and compare its performance with *Coq* [14], *Twelf* [13] and *Maude* [9].

2 The λΠ-calculus modulo

2.1 Language Description

In this section we describe the λΠ-calculus modulo, a variant of the λ-calculus with dependent types (λΠ-calculus) where β-reduction is extended with user-defined rewrite rules. The syntax of the λΠ-calculus modulo for terms, rewrite rules and contexts is given in Figure 1. A term is either a variable, an application, a λ-abstraction, a product type or a sort (either **Type** or **Kind**). A typing context is a list of declarations, *i.e.*, pairs of a variable name and a term (its type). A rewrite rule is a triple made of a typing context and two terms. Finally a context is a list of declarations and rewrite rules.

As contexts may contain rewrite rules, they define a rewriting system. Given a context Γ we write $\to_\Gamma$ the smallest relation, compatible with the structure of terms, such that for any rule $[\Delta]\ l \hookrightarrow r$ in Γ and substitution σ with $dom(\sigma) = dom(\Delta)$, $\sigma l \to_\Gamma \sigma r$. We note $\to_{\beta\Gamma}$ for $\to_\beta \cup \to_\Gamma$ and $\equiv_{\beta\Gamma}$ for the congruence generated by $\to_{\beta\Gamma}$.

x	$\in$	$\mathcal{V}$ (an infinite set)	(Variable)
s	::=	**Type** | **Kind**	(Sort)
t, u, A, B	::=	$x \mid t\ u \mid \lambda x^A.t \mid \Pi x^A.B \mid s$	(Term)
Δ	::=	$\emptyset \mid \Delta(x:t)$	(Typing Context)
$\mathcal{R}$	::=	$[\Delta]\ t \hookrightarrow u$	(Rewrite Rule)
Γ	::=	$\emptyset \mid \Gamma(x:t) \mid \Gamma\mathcal{R}$	(Context)

Figure 1: Syntax of the λΠ-calculus modulo.

The inference rules for the λΠ-calculus modulo, defining well-formed contexts and well-typed terms, are given in Figure 2. One can notice that these rules differ from the rules of λΠ-calculus only in two points. First, the congruence in the **Conv** rule is extended from $\equiv_\beta$ to $\equiv_{\beta\Gamma}$, allowing for more terms to be typed. Secondly, there is a new rule **Rw** which makes possible the addition of rewrite rules in the context, thus extending explicitly the considered rewrite system. This latter rule is a novelty with respect to previous formalizations. Addition of rewrite rules becomes an iterative process: rules previously added can be used to type new rules. We obtain this way a formalism that is close to its actual implementation in *Dedukti* where a rewrite rule can appear anywhere in the specification.

$$\textbf{(Empty)}\ \frac{}{\emptyset\ wf} \qquad \textbf{(Dec)}\ \frac{\Gamma\ wf \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma(x:A)\ wf}$$

$$\textbf{(Rw)}\ \frac{\Gamma\ wf \qquad \forall \sigma \in \mathcal{S}(\Gamma \vdash \sigma l : A \Rightarrow \Gamma \vdash \sigma r : A)}{\Gamma([\Delta] l \hookrightarrow r)\ \text{wf}} \text{ with } \mathcal{S} := \{\sigma | dom(\sigma) = \Delta \textit{ and } (x:A) \in \Delta \Rightarrow \Gamma \vdash \sigma x : \sigma A\}$$

$$\textbf{(Type)}\ \frac{\Gamma\ wf}{\Gamma \vdash \textbf{Type} : \textbf{Kind}} \qquad \textbf{(Var)}\ \frac{\Gamma\ wf \qquad (x:A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\textbf{(App)}\ \frac{\Gamma \vdash t : \Pi x^A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B[x \backslash u]} \qquad \textbf{(Conv)}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}$$

$$\textbf{(Abs)}\ \frac{\Gamma \vdash A : \textbf{Type} \qquad \Gamma(x:A) \vdash t : B \qquad B \neq \textbf{Kind}}{\Gamma \vdash \lambda x^A.t : \Pi x^A.B}$$

$$\textbf{(Prod)}\ \frac{\Gamma \vdash A : \textbf{Type} \qquad \Gamma(x:A) \vdash B : s}{\Gamma \vdash \Pi x^A.B : s}$$

Figure 2: Typing rules for λΠ-calculus modulo.

2.2 Typing Rewrite Rules

When adding a rewrite rule in the context one must check that it is compatible with typing. More precisely we want the so-called Subject Reduction property to hold: the type of a term should be invariant by rewriting.

Subject Reduction: if $\Gamma \vdash t : T$ and $t \to_{\beta\Gamma} t'$ then $\Gamma \vdash t' : T$.

Unexpectedly, Subject Reduction for $\to_\beta$ might fail in the presence of rewrite rules since one has to prove that $\Pi x^{A_1}.B_1 \equiv_{\beta\Gamma} \Pi x^{A_2}.B_2$ implies $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$. However this is provable if $\to_{\beta\Gamma}$ is confluent since, in this case, both Π-types reduce to a common term $\Pi x^A.B$ which witnesses that $A_1 \equiv_{\beta\Gamma} A \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B \equiv_{\beta\Gamma} B_2$. Subject reduction for $\to_\Gamma$ is usually ensured by requiring rewrite rules $[\Delta]\ l \hookrightarrow r$ to enjoy the following property.

Condition 1: There exists T such that $\Gamma\Delta \vdash l : T$ and $\Gamma\Delta \vdash r : T$.

This condition is often found in the literature, including our previous work on *Dedukti* [5], but is actually too strong. Figure 3 shows an example where a rewrite rule does not verify the condition above while preserving subject reduction. Indeed, on the last line, the term **Last** n_1 (**Cons** n_2 a l) is well-typed if and only if $n_1 \equiv_{\beta\Gamma} S\ n_2$, which is trivially false because n_1 and n_2 are variables. However we know that, if for some terms t_1 and t_2, the left-hand side of the rule **Last** t_1 (**Cons** t_2 a l) is well-typed then we have $t_1 \equiv_{\beta\Gamma} S\ t_2$ and the right-hand side has the same type. Thus the Subject Reduction property is preserved. Of course, we could have defined the rule as $[(n : Nat)(a : A)(l : Listn\ n)]$ **Last** (**S** n) (**Cons** n a l) $\hookrightarrow$ **Last** n l, which meets the previous condition. But we would have introduced non-linearity, resulting in less efficient rewriting. Indeed, now a conversion test is needed to check if a term is rewritable. Moreover non-linear rewrite systems are more complex to study when it comes to proving confluence and strong normalization.

In [4] Blanqui remarked that for a rewrite rule $[\Delta]\ l \hookrightarrow r$ a weaker condition is actually sufficient (and necessary) to ensure the subject reduction property.

Condition 2: For any substitution $\sigma \in \mathcal{S}$ (*i.e.* $dom(\sigma) = \Delta$ and $(x : A) \in \Delta$ implies $\Gamma \vdash \sigma x : \sigma A$), if $\Gamma \vdash \sigma l : A$ then $\Gamma \vdash \sigma r : A$.

Our previous example verifies this condition. Thus this solution allows more rules to be added in a context and avoids unnecessary non linear rules. The downside of this approach is that the condition becomes undecidable. However it is easy to prove in many cases. For instance one might find by unification a substitution ρ such that if $\sigma \in \mathcal{S}$ then $\sigma = \sigma_0 \circ \rho$ and test if $\Gamma\Delta' \vdash \rho l : T$ implies $\Gamma\Delta' \vdash \rho r : T$ with $\Delta' := \Delta \backslash (dom(\rho))$.

$(Nat : \mathbf{Type})$
$(Z : Nat)$
$(S : \Pi x^{Nat}.Nat)$

$(A : \mathbf{Type})$
$(Listn : \Pi x^{Nat}.\mathbf{Type})$
$(Nil : ListnZ)$
$(Cons : \Pi a^{A}.\Pi n^{Nat}.\Pi x^{(Listn\ n)}.Listn\ (S\ n))$

$(Last : \Pi n^{Nat}.\Pi x^{(Listn\ n)}.A)$

$[(n_1 : Nat)(n_2 : Nat)(a : A)]$
$\quad Last\ n_1\ (Cons\ n_2\ a\ Nil) \hookrightarrow a$
$[(n_1 : Nat)(n_2 : Nat)(a : A)(l : List\ n_2)]$
$\quad Last\ n_1\ (Cons\ n_2\ a\ l) \hookrightarrow Last\ n_2\ l$

```
(; Peano Integers ;)
Nat: Type.
Z: Nat.
S: Nat -> Nat.
(; Lists of size n ;)
A:Type.
Listn: Nat->Type.
Nil: Listn Z.
Cons: A->n:Nat->Listn n->Listn (S n).
(; Last element of a list ;)
Last: n:Nat -> Listn n -> A.
(; Rewrite Rules for Last ;)
[n1:Nat;n2:Nat;a:A]
      Last n1 (Cons n2 a Nil) --> a
[n1:Nat;n2:Nat;a:A;l:List n2]
      Last n1 (Cons n2 a l) --> Last n2 l.
```

Figure 3: The function **Last** in the λΠ-calculus modulo and the corresponding source in *Dedukti*.

3 Implementation

3.1 General Description

Our current version of the type-checker for the λΠ-calculus modulo is a complete rewrite of the previous version of *Dedukti* [5]. It is now entirely written in *OCaml* and consists in about a thousand lines of code. It implements the standard algorithm for type checking semi-full *Pure Type Systems* [15] as well as a term reduction machine inspired from *Matita*'s [1] adapted to rewrite rules.

Dedukti is released under the *CeCILL-B* license. Its source code is available at `https://www.rocq.inria.fr/deducteam/Dedukti`.

3.2 Benchmarks

We assessed the efficiency of *Dedukti* for both type-checking and rewriting. We compared the results with those of *Coq* (version 8.3pl4), a proof assistant for the calculus of inductive constructions, *Twelf* (version 1.7.1+), an implementation of the λΠ-calculus, and *Maude* (version 2.6), a high-level language which supports rewriting logic computation. The tests were run on a Linux laptop with a processor Intel Core i7-3520M CPU @ 2.90GHz x 4 and 16GB of Ram. The different benchmark files can be obtained on *Dedukti*'s website.

3.2.1 *Dedukti* vs. *Coq* vs. *Twelf*

We chose to use the *OpenTheory* [12] library as a benchmark to assess type-checking performance. This library was conceived as a way to share theorems between theorem provers implementing higher-order logic. Its core is composed of more than 80 files of variable size (up to 250Mo) containing general definitions and theorems. Thanks to *Holide* [2], the *OpenTheory* format can be encoded into the λΠ-calculus (*i.e.*, without rewrite rule) and printed in *Coq*, *Twelf* or *Dedukti*'s input format. It can also be translated, via an encoding similar to [10] and using one rewrite rule, in the λΠ-calculus modulo and printed in *Dedukti*'s input format. The latter translation is not available for *Coq* or *Twelf* as these tools cannot deal with user-defined rewrite rules. Some of the results are given Table 4. We can see that *Dedukti* is globally 20% faster than *Coq* and 30% faster than *Twelf* on the λΠ-calculus encoding. On the other hand type checking the λΠ-calculus modulo translation is more than 20 times faster. The explanation is that rewrite rules permit to obtain smaller terms that moreover have less type dependencies thus requiring less conversion tests while type checking.

File	**Size**	***Dedukti*** (λΠ)	***Coq*** (λΠ)	***Twelf*** (λΠ)	***Dedukti*** (λΠm)
natural-exp-thm.dk	55M	11	13	20	1
list-def.dk	84M	21	20	32	1
set-thm.dk	97M	30	63	33	2
relation-natural-thm.dk	122M	53	40	56	2
real-def.dk	259M	53	107 (1mn47)	-	3
All files (88)	1.4G	395 (6mn35)	470 (7mn50)	523 (8mn43)	17

Figure 4: Type-checking time (in seconds).

3.2.2 *Dedukti* vs *Maude*

As a preliminary test of *Dedukti*'s rewriting capabilities, we ran both *Dedukti* and *Maude* on a series of handwritten benchmarks consisting on computations involving arithmetic functions defined as algebraic rewrite rules on unary integers. Results are given in Table 5. These tests show no clear correspondence between time spent by *Dedukti* and *Maude*. Further work should be done to investigate the differences between the rewrite strategy used by both tools.

4 Conclusion

We have presented a version of the λΠ-calculus modulo where rewrite rules are explicitly added in contexts making their addition an iterative process where previous rules can be used to type

Expression	***Dedukti***	***Maude***	**Expression**	***Dedukti***	***Maude***
2^{10}	31	6	2^{11}	267 (4mn27)	45
3^6	5	1	3^7	174 (2mn54)	28
$5 * 4^5$	56	53	$10 * 4^5$	120 (2mn)	218 (3mn38)
$(10 * 10) * (10 * 10)$	4	54	$10 * (10 * (10 * 10))$	1	16

Figure 5: Rewriting time (in seconds).

the new ones. We have discussed the condition rewrite rules must satisfy in order to preserve the Subject Reduction property and we have given a criterion weaker than the usual one. We have described the new version of *Dedukti*, and we have shown that it is faster than *Coq* and *Twelf*. We have also given preliminary comparison data with *Maude*.

At present our type-checking algorithm assumes the confluence (for completeness) and the strong normalization (for termination) of $\rightarrow_{\beta\Gamma}$. As only algebraic linear rewrite rules are implemented in *Dedukti*, the confluence of $\rightarrow_{\beta\Gamma}$ follows from the confluence of $\rightarrow_{\Gamma}$ [16]. Regarding the strong normalization of the combination of β-reduction with an arbitrary rewrite system, general critera exist in the literature. See for instance [3] for object-level rewriting or [4] for object-level and type-level rewriting. We plan to integrate these works into *Dedukti* in a future release.

References

[1] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. A Compact Kernel for the Calculus of Inductive Constructions. *Sadhana*, 2009.

[2] A. Assaf and G. Burel. Holide: `https://www.rocq.inria.fr/deducteam/Holide/`.

[3] G. Barthe and H. Geuvers. Modular Properties of Algebraic Type Systems. In *HOA*, 1995.

[4] F. Blanqui. Definitions by rewriting in the Calculus of Constructions. *MSCS*, 2005.

[5] M. Boespflug, Q. Carbonneaux, and O. Hermant. The λΠ-calculus Modulo as a Universal Proof Language. In *PxTP*, 2012.

[6] N.G. Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*. Springer Berlin Heidelberg, 1970.

[7] G. Burel. A Shallow Embedding of Resolution and Superposition Proofs into the λΠ-Calculus Modulo. In *PxTP*, 2013.

[8] R. Cauderlier. Focalide: `https://www.rocq.inria.fr/deducteam/Focalide`.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *RTA*, 2003.

[10] G. Dowek D. Cousineau. Embedding Pure Type Systems in λΠ-Calculus Modulo. In *TLCA*, 2007.

[11] D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In *LPAR*, 2013.

[12] J. Hurd. The OpenTheory Standard Theory Library. In *NASA Formal Methods*, 2011.

[13] F. Pfenning and C. Schürmann. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In *CADE*, 1999.

[14] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA. Version 8.3, available at `http://coq.inria.fr/refman/index.html`.

[15] L. S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for Pure Type Systems. In *Types for Proofs and Programs*. Springer Berlin Heidelberg, 1994.

[16] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1994.